COL 380 : A1

Sidharth Agarwal - 2019CS50661

January 16, 2022

Profiling:

So for profiling I majorly focused on valgrind. After compiling the classify.cpp, run the following commands in the order given for running 1 iteration over 4 threads.

- *valgrind –tool=cachegrind –cachegrind-out-file=out ./classify rfile dfile 100907*2 *4 *1*
- *cg_annotate out /home/sidharth/Parallel380/A1/classify.cpp > profile.txt*

```
--------------------------------------------------------------------------------
-- User-annotated source: /home/sidharth/Parallel380/A1/classify.cpp
--------------------------------------------------------------------------------
Ir            I1mr ILmr Dr            D1mr        DLmr        Dw        D1mw    DLmw

         .    .    .           .           .           .         .       .      .  #include "classify.h"
         .    .    .           .           .           .         .       .      .  #include <omp.h>
         .    .    .           .           .           .         .       .      .
         .    .    .           .           .           .         .       .      .  Data classify(Data &D, const Ranges &R, unsigned int numt)
        12    2    2           1           0           0         7       0      0  { // Classify each item in D into intervals (given by R). Finally, produce in D2 data sorted by interval
         2    0    0           0           0           0         0       0      0     assert(numt < MAXTHREADS);
     3,090    3    3          17          15          15         1       1      1     Counter counts[R.num()]; // I need on counter per interval. Each counter can keep pre-thread subcount.
         .    .    .           .           .           .         .       .      .
         .    .    .           .           .           .         .       .      .     /*
         .    .    .           .           .           .         .       .      .     #pragma omp for
         .    .    .           .           .           .         .       .      .     for(int i=0;i<D.ndata;i+=1)
         .    .    .           .           .           .         .       .      .     {
         .    .    .           .           .           .         .       .      .        int tid = omp_get_thread_num(); // I am thread number tid
         .    .    .           .           .           .         .       .      .        int v = D.data[i].value = R.range(D.data[i].key);
         .    .    .           .           .           .         .       .      .        counts[v].increase(tid);
         .    .    .           .           .           .         .       .      .     }
         .    .    .           .           .           .         .       .      .     */
         .    .    .           .           .           .         .       .      .     //*
        71    2    2          33           9           6        25       1      0     #pragma omp parallel num_threads(numt)
         .    .    .           .           .           .         .       .      .     {
         4    0    0           0           0           0         4       0      0        int tid = omp_get_thread_num(); // I am thread number tid
 4,036,300    0    0   1,009,080           1           0         0       0      0        for(int i=tid; i<D.ndata; i+=numt) { // Threads together share-loop through all of Data
 4,036,300    1    1   1,009,080     504,538     478,185 1,009,072       4      1           int v = D.data[i].value = R.range(D.data[i].key);// For each data, find the interval of data's key,
         .    .    .           .           .           .         .       .      .                                   // and store the interval id in value. D is changed.
 1,009,072    0    0   1,009,072       4,004       3,036         0       0      0           counts[v].increase(tid); // Found one key in interval v
         .    .    .           .           .           .         .       .      .        }
         .    .    .           .           .           .         .       .      .     }
         .    .    .           .           .           .         .       .      .     //*/
         .    .    .           .           .           .         .       .      .
         .    .    .           .           .           .         .       .      .     // Accumulate all sub-counts (in each interval;'s counter) into rangecount
         7    0    0           1           0           0         1       0      0     unsigned int *rangecount = new unsigned int[R.num()];
     2,012    1    1           2           1           1         0       0      0     for(int r=0; r<R.num(); r++) { // For all intervals
     1,001    0    0           0           0           0     1,001      62     61        rangecount[r] = 0;
    15,015    0    0       2,002       1,000         247         0       0      0        for(int t=0; t<numt; t++) // For all threads
     4,004    0    0       4,004           0           0         0       0      0           rangecount[r] += counts[r].get(t);
         .    .    .           .           .           .         .       .      .        // std::cout << rangecount[r] << " elements in Range " << r << "\n"; // Debugging statement
         .    .    .           .           .           .         .       .      .     }
         .    .    .           .           .           .         .       .      .
         .    .    .           .           .           .         .       .      .     // Compute prefx sum on rangecount.
     2,006    0    0           0           0           0         0       0      0     for(int i=1; i<R.num(); i++) {
     2,000    0    0       2,000          49           0         0       0      0        rangecount[i] += rangecount[i-1];
         .    .    .           .           .           .         .       .      .     }
```

For gprof simply use the command *gprof classify gmon.out ¿ analysis.txt* and we will get the profiling.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
100.27      8.32     8.32                             readRanges(char const*)
  0.00      8.32     0.00        6     0.00     0.00  classify(Data&, Ranges const&, unsigned int)
  0.00      8.32     0.00        6     0.00     0.00  timedwork(Data&, Ranges const&, unsigned int)
  0.00      8.32     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z8classifyR4DataRK6Rangesj
  0.00      8.32     0.00        1     0.00     0.00  _GLOBAL__sub_I__Z9timedworkR4DataRK6Rangesj

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.
```

Initial Analysis:

So we carefully look at the profile files of this program, we can easily see that the majority of cache misses occur in inner for loops of both the parallelised code sections. More specifically these four lines of codes *int v = D.data[i].value = R.range(D.data[i].key)*, *if(D.data[d].value == r)*, *D2.data[rangecount[r-1]+rcount++] = D.data[d];*.

Approaches Tested:

rFactor:

So now we can see that there is rangecounter array which is present in the 2nd parallel loop, this results in a lot of cache misses. So a probable reason for that could be false memory sharing. Since the same rangecount chunk of array gets loaded in each of the threads, so we can instead add padding to this array, to avoid false sharing. Now unfortunately this approach doesn't lead to a lot of reduction in time and is only able to reduce time from 430ms to 400ms on average. Similarly I also tested increasing the size of Counter object from 32 to 1028 and padding the counts array as well and it still didn't give a reduction in time nor reduction in cache misses.

Parallel for

So in first loop we are using omp parallel and assigning the data to each thread based on the modulus of the data item, so I decided to test by using parallel for instead so that all the neighbouring elements of Data array are in the same thread. Like sort of a partition of elements among the threads. Unfortunately the approach resulted in increase in time from around 430 ms to 520ms.

Partition over modulus

As we can currently observe that in the first parallel computation we divide the data across different threads based on the modulus of the data point, now to avoid false sharing memory we could instead parition the whole data into continous segments and then assign it to different threads. Unfortunately it doesn't also lead to a reduction in time.

Algorithm of second parallel loop:

So we can observe that the second parallelised loop is first traversing over all the elements of Ranges and then tested whether each Data element lies in that range. This is highly inefficient since we already know where exactly each Data element lies. Hence instead we can simply iterate over all the elements of Data and add them to their respective bins(Ranges) will keeping a track of how many total elements eventually will be there in that bin and how many have been added yet. This resulted in a decrease in time from around 430ms to 220ms.
Note: This was not improving parallelism, but still mentioning an approach I worked on.